Having Fun With 31.521 Shell Scripts

Nicolas Jeannerod, Yann Régis-Gianas, Ralf Treinen

Having Fun With 31.521 Shell Scripts

Nicolas Jeannerod
Département d'Informatique
École normale supérieure
Paris, France
Email: nicolas.jeannerod@irif.fr

Yann Régis-Gianas
IRIF
Université Paris-Diderot, CNRS
INRIA PI.R2, Paris, France
Email: yrg@irif.fr

Ralf Treinen
IRIF
Université Paris-Diderot, CNRS
Paris, France
Email: treinen@irif.fr

***Abstract*—Statically parsing shell scripts is, due to various peculiarities of the shell language, a challenge. One of the difficulties is that the shell language is designed to be executed by intertwining reading chunks of syntax with semantic actions.**

We have analyzed a corpus of 31.521 POSIX shell scripts occurring as maintainer scripts in the Debian GNU/Linux distribution. Our parser, which makes use of recent developments in parser generation technology, succeeds on 99.9% of the corpus. The architecture of our tool allows us to easily plug in various statistical analyzers on the syntax trees constructed from the shell scripts. The statistics obtained by our tool are the basis for the definition of a model which we plan to use in the future for the formal verification of scripts.

I. Introduction

Free and open-source software (FOSS) provides today a gigantic corpus of software which is available for everyone for use, modification, and scrutiny. FOSS is nowaday ubiquitous in almost everybody's computing environment. The probably most important way of acquiring, installing, and updating a FOSS-based system is by using one of the so-called distributions [7]. Debian is one of the oldest distributions of GNU/Linux systems, and certainly the largest by the number of software packages maintained by the distribution itself.

A software package deployed by such a distribution consists, roughly speaking, of an archive of files which are to be placed on the target machine, meta-data like dependencies on other packages, and possibly scripts (called *maintainer scripts* in the context of Debian) which are to be executed at various stages of package installation, removal, and upgrade. As part of our long-term goal of applying formal methods to the quality assessment of free and open-source software (see, e.g., [13], [4], [1]), we are currently trying to apply methods of formal program verification to the maintainer scripts that are part of the Debian GNU/Linux distribution.

Errors occurring during an attempt to install a package can be classified into static and dynamic [6]. Static failures, which are due to unsatisfiable meta-data of packages such as dependencies, conflicts, etc., are usually detected by a meta-installer like APT even before any actual installation process in started. Dynamic failures, however, are due to failures of maintainer scripts. Failures may also go undetected, that is the installation process terminates without reporting a failure, even though the resulting system is in an inconsistent state. We hope to be able in the future to address these dynamic failures, and possibly even undetected failures.

As a first step towards the goal of verifying maintainer scripts we have defined an intermediate language [17] , which can be used as input to the verification tools and which avoids some of the pitfalls of POSIX shell. The language is designed with the goal that an automatic translation of actual maintainer scripts written in POSIX shell into that language is possible. The design of this language hinges on an analysis on what features of the POSIX shell language, and what UNIX commands with what options are actually encountered in the corpus containing the 31.832 maintainer scripts to be found in Debian sid. We are interested in questions such as: Do scripts use EVAL? Are recursive functions actually used? How are WHILE loops used in scripts? Are the lists over which FOR loops iterate known statically? What options of UNIX commands like LN are mostly used, and in what combination?

Our aim is to have a tool which allows us to define easily new kinds of analyses on the corpus. In order to do this, we need a way of constructing concrete syntax trees for all scripts of our corpus. New analyses which are executed on these concrete syntax trees are then easily written. Static syntax analysis of shell scripts, however, is far from trivial, and even impossible in some corner cases as we will explain in the next section. One, but not the only obstacle is the fact that the shell language was designed to be read by an interpreter on the fly, intertwined with various expansion mechanisms. Another problem is that the shell language is a serious challenge for the textbook approach of pipelining lexing and parsing. Nevertheless, we managed to write a modular syntactic analyzer, largely using code generators, by exploiting recent developments in parser generation technology.

The rest of this article is organized as follows: We start by showing the difficulties that we have encountered with the syntax of POSIX shell in Section II; Section III explains how we could maintain a modular, though nonstandard, design of our syntactic analyzer despite the pitfalls of shell language. In Section IV we sketch how different analyzers of shell scripts can be written and combined with the syntactic analyzer. Some statistical results obtained on the corpus of Debian maintainer scripts are given in Section V. We conclude with some current limitations of our tool and plans for future work in Section VI, and compare our approach to related work in Section VII.

II. The perils of POSIX shell

The POSIX Shell Command Language is specified by the Open Group and IEEE in the volume "Shell & Utilities" of the POSIX standard. Our implementation is based on the latest published draft of this standard [12].

This standardization effort synthesizes the common concepts and mechanisms that can be found in the most common implementations of shell interpreter (like `bash` or `dash`). Unfortunately, it is really hard to extract a high-level declarative specification out of these existing implementations because the shell language is inherently irregular, and because its unorthodox design choices fit badly in the usual specification languages used by other programming language standards.

Syntactic analysis is most often decomposed into two distinct phases: (i) *lexical analysis*, which synthesizes a stream of lexemes out of a stream of input characters by recognizing lexemes as meaningful character subsequences and by ignoring insignificant character subsequences such as layout; (ii) *parsing* which synthesizes a parse tree from the stream of lexemes according to some formal grammar.

In this section, we describe several aspects which make the shell language hard (and actually impossible in general) to parse using the standard decomposition described above, and more generally using the standard parsing tools and techniques. These difficulties not only raise a challenge in terms of programming but also in terms of reliability. Indeed, the standard techniques to implement syntactic analyzers are based on code generators. These tools take as input high-level formal descriptions of the lexical conventions and of the grammar and produce low-level efficient code using well-understood computational devices (typically finite-state transducers for lexical analysis, and pushdown automata for parsing). This standard approach is trustworthy because (i) the high-level descriptions of the lexical conventions and grammar are very close to their counterparts in the specification; (ii) the code generators are usually based on well-known algorithms like LR-parsing which have been studied for almost fifty years. Despite of the pitfalls of the shell language, we nonetheless managed to maintain an important part of generated code in our implementation, described in Section III.

A. Non standard specification of lexical conventions

1) Token recognition: In usual programming languages, most of the categories of tokens are specified by means of regular expressions. As explained earlier, lexer generators (e.g. `lex`) conveniently turn such high-level specifications into efficient finite state transducers, which makes the resulting implementation both reliable and efficient.

The token recognition process for the shell language is described in Section 2.3 of the specification [12], unfortunately without using any regular expressions. While other languages use regular expressions with a longest-match strategy to delimit the next lexeme in the input, the specification of the shell language uses a state machine which explains instead how tokens must be delimited in the input and how the delimited chunks of input must be classified into two categories: words and operators.

The state machine which recognizes the tokens is unfortunately not a regular finite state machine. It is almost as powerful as a pushdown automaton since it must be able to recognize nested quotations like the ones found in the following example.

Example 1 (Quotations): Consider the following input:

```
1 BAR='foo'"ba"r
2 X=0 echo x$BAR" "$(echo $(date))
```

By the lexical conventions of most programming languages, the first line would be decomposed as five distinct tokens (namely `BAR`, `=`, `'foo'`, `"ba"` and `r`) while the lexical conventions of the shell language considers the entire line `BAR='foo'"ba"r` as a single token, classified into the category of words. On the second line, the input is split into the tokens `X=0`, `echo` and `x$BAR" "$(echo $(date))`. Notice that the third token contains nested quotations of the form `$(..$(..) )` the recognition of which is out of the scope of regular finite state machines (without a stack).

2) Layout: The shell language also has some unconventional lexical conventions regarding the interpretation of newline characters. Usually, newline characters are simply ignored by the lexing phase since they only serve as delimiters between tokens. In shell, however, newline characters are meaningful, and there even are four different interpretations of a newline depending on the parsing context. Therefore, most of the newline characters (but not all, as we shall see in the next example) must be transmitted to the parser.

Example 2 (Interpretations of newline characters): The four interpretations of the newline characters occur in the following example:

```
1 $ for i in 0 1
2 > # Some interesting numbers
3 > do echo $i \
4 > + $i
5 > done
```

On line 1, the newline character has a syntactic meaning because it acts as a marker for the end of the sequence over which the **for**-loop is iterating. On line 2, the newline character at the end of the comment must not be ignored. But here, it is collapsed with the newline character of the previous line. On line 3, the newline character is preceded by a backslash. This sequence of characters is interpreted as a line-continuation, which must be handled at the lexing level. Therefore in that case, the newline is actually interpreted as layout. On lines 4 and 5, the final newline terminates a command.

3) Here-documents: Depending on the parsing context, the lexer must switch to a special mode to deal with here-documents. Here-documents are chunks of text embedded in a shell script. They are commonly used to implement some form of template-based generation of files (since they may contain variables). To use that mode, the user provides textual end-markers and the lexer then interprets all the input up to an end-marker as a single token of the category of words. The input characters are copied verbatim into the representation

of the token, with the possible exception of quotations which may still be recognized exactly as in the normal lexing mode.

Example 3 (Here-documents):

```
1 cat > notifications << EOF
2 Hi $USER,
3 Enjoy your day!
4 EOF
5 cat > toJohn << EOF1 ; cat > toJane << EOF2
6 Hi John!
7 EOF1
8 Hi Jane!
9 EOF2
```

In this example, the text on lines 2 and 3 is interpreted as a single word which is passed as input to the `cat` command. The first `cat` command of line 5 is fed with the content of line 6 while the second `cat` command of line 6 is fed with the content of line 8. This example with two successive here-documents illustrates the non-locality of the lexing process of here-document: the word related to the end-marker `EOF1` is recognized several tokens after the introduction of `EOF1`. This non-locality forces some form of forward declaration of tokens, the contents of which is defined afterwards.

B. Parsing-dependent lexical analysis

While the recognition of tokens is independent from the parsing context, their classification into words, operators, newlines and end-of-files must be refined further to obtain the tokens actually used in the formal grammar specified by the standard. The declaration of these tokens is reproduced in Figure 1. While a chunk categorized as an operator is easily transformed into a more specific token like `AND_IF` or `OR_IF`, an input chunk categorized as a word can be promoted to a reserved word or to an assignment word only if the parser is expecting such a token at the current position of the input; otherwise the word is not promoted and stays a `WORD`. This means that the lexical analysis has to depend on the state of the parser.

1) Parsing-sensitive word assignment recognition: The promotion of a word to an assignment depends both on the position of this word in the input and on the string representing that word. The string must be of the form `w=u` where the substring `w` must be a valid name, a lexical category defined in Section 3.235 of the standard by the following sentence:

> [...] a word consisting solely of underscores, digits, and alphabetics from the portable character set. The first character of a name is not a digit.

Example 4 (Promotion of a word as an assignment):

```
1 CC=gcc make
2 make CC=cc
3 ln -s /bin/ls "X=1"
4 "./X"=1 echo
```

On line 1, the word `CC=gcc` is recognized as a word assignment of `gcc` to `CC` because `CC` is a valid name for a variable, and because `CC=gcc` is written just before the command name of the simple command `make`. On line 2, the word `CC=cc` is not promoted to a word assignment because it appears after the command name of a simple command. On line 4, since `"./X"` is not a valid name for a shell variable, the word `"./X=1"` is not promoted to a word assignment and is interpreted as the command name of a simple command.

```
%token  WORD
%token  ASSIGNMENT_WORD
%token  NAME
%token  NEWLINE
%token  IO_NUMBER

/* The following are the operators (see XBD Operator)
   containing more than one character. */
%token  AND_IF    OR_IF    DSEMI
/*      '&&'      '||'     ';;'    */
%token  DLESS  DGREAT  LESSAND
/*      '<<'   '>>'    '<&'     */
%token GREATAND  LESSGREAT  DLESSDASH
/*     '>&'      '<>'       '<<-'   */
%token  CLOBBER
/*      '>|'   */
/* The following are the reserved words. */
%token  If    Then    Else    Elif    Fi    Do    Done
/*      'if'  'then'  'else'  'elif'  'fi'  'do'  'done' */
%token  Case    Esac    While    Until    For
/*      'case'  'esac'  'while'  'until'  'for' */
/* These are reserved words, not operator tokens, and are
   recognized when reserved words are recognized. */
%token  Lbrace    Rbrace    Bang
/*      '{'       '}'       '!'   */
%token  In
/*      'in'   */
```

Fig. 1. The tokens of the shell language grammar.

2) Parsing-sensitive keyword recognition: A word is promoted to a reserved word if the parser state is expecting this reserved word at the current point of the input:

Example 5 (Promotion of a word to a reserved word):

```
1 for i in a b; do echo $i; done
2 ls for i in a b
```

On line 1, the words `for`, `in`, `do`, `done` are recognized as reserved words. On line 2, they are not recognized as such since they appear in position of command arguments for the command `ls`.

C. Evaluation-dependent lexical analysis

The lexical analysis also depends on the evaluation of the shell script. Indeed, the `alias` builtin command of the POSIX shell amounts to the dynamic definition of macros which are expanded just before lexical analysis. Therefore, even the lexical analysis of a shell script cannot be done without executing it, that is lexical analysis of unrestricted shell scripts is undecidable. Fortunately, restricting the usage of the `alias` command to only top level commands, that is outside of control structures, restores decidability.

Example 6 (Lexical analysis is undecidable):

```
1 if ./foo; then
2   alias x="ls"
3 else
4   alias x=""
5 fi
6 x for i in a b; do echo $i; done
```

To decide if `for` is a reserved word, a lexer must be able to decide the success of an arbitrary program `./foo`, which is impossible. Hence, the lexer must wait for the evaluation of the first command to be able to parse the second one.

Another problematic feature of the shell language is `eval`. This builtin constructs a command by concatenating its arguments, separated by spaces, and then executes the constructed command in the shell. In other words, the *construction* of the command that will be executed depends on the execution of the script, and hence cannot be statically known by the parser.

D. Ambiguous grammar

The grammar of the shell language is given in Section 2.10 of the standard. At first sight, the specification seems to be written in the input format of the YACC parser generator. Alas, YACC cannot be given this specification as-is for two reasons: (i) the specification is annotated with nine special rules which are not directly expressible in terms of YACC's parsing mechanisms ; (ii) the grammar contains LR(1) conflicts.

1) Special rules: The nine special rules of the grammar are actually the place where the parsing-dependent lexical conventions are explained. By lack of space, we only focus on the Rule 4 to give the idea. This is an excerpt from the standard describing this rule:

> *[Case statement termination]*
> *When the* **TOKEN** *is exactly the reserved word* **esac***, the token identifier for* **esac** *shall result. Otherwise, the token* **WORD** *shall be returned.*

The grammar refers to that rule in the following case:

```
pattern:
 WORD /* Apply rule 4 */
| pattern '|' WORD /* Do not apply rule 4 */
;
```

Roughly speaking, this annotation says that when the parser is recognizing a `pattern` and when the next token is the specific `WORD` **esac**, then the next token is actually not a `WORD` but the token `Esac`. In that situation, the LR parser must pop up its stack to a state where it is recognizing the non terminal `case_clause` defined as follows:

```
case_clause:
Case WORD linebreak in linebreak case_list Esac
| Case WORD linebreak in linebreak case_list_ns Esac
| Case WORD linebreak in linebreak Esac
```

to end the recognition of the current `case_list`.

2) LR(1) conflicts: Our LR(1) parser generator detects five shift/reduce conflicts in the YACC grammar of the standard. All these conflicts are related to the analysis of newline characters in the body of case items in case analysis. Indeed, the grammar is not LR(1) with respect to the handling of these newline characters. Here is the fragment of the grammar to be incriminated for these conflicts:

```
compound_list: linebreak term
             | linebreak term separator
case_list_ns : case_list case_item_ns
             |           case_item_ns
             ;
case_list    : case_list case_item
             |           case_item
             ;
case_item_ns :     pattern ')' linebreak
             |     pattern ')' compound_list
             | '(' pattern ')' linebreak
             | '(' pattern ')' compound_list
             ;
case_item    :     pattern ')' linebreak     DSEMI linebreak
             |     pattern ')' compound_list DSEMI linebreak
             | '(' pattern ')' linebreak     DSEMI linebreak
             | '(' pattern ')' compound_list DSEMI linebreak
separator    : separator_op linebreak
             | newline_list
             ;
```

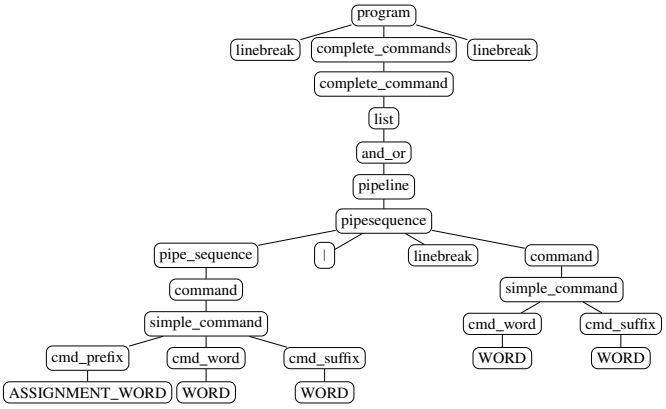

Fig. 2. Concrete syntax tree for `CC=gcc make -C . | grep 'error'`

```
newline_list : NEWLINE
             | newline_list NEWLINE
             ;
linebreak    : newline_list
             | /* empty */
             ;
```

When a `NEWLINE` is encountered after `term` in a context of the following form:

```
case ... in ...)
```

an LR parser cannot choose between reducing the `term` into a `compound_list` or shifting the `NEWLINE` to start the recognition of the final `separator` of the current `compound_list`.

Fortunately, as the newline character has no semantic meaning in the shell language, choosing between reduction or shift has no significant impact on the output parse tree.

III. UNORTHODOX PARSING

As explained in the introduction, the purpose of our shell script analyzer is to validate several hypotheses about the *idioms* used by package maintainers when writing maintainer scripts. We may *a priori* be interested in any kind of question on the way shell scripts are written, hence the parser should not do any abstraction on the syntax. Therefore, contrary to parsers typically found in compilers or interpreters, our parser does not produce an abstract syntax tree from a syntactically correct source but a parse tree instead. A parse tree, or concrete syntax tree, is a tree whose nodes are grammar rule applications. Figure 2 gives an example of such a concrete syntax tree. Because we need concrete syntax trees (and also, as we shall see, because we want high assurance about the compliance of the parser with respect to the POSIX standard), reusing an existing parser implementation was not an option.

As a consequence of the difficulties explained in Section II, we cannot write a syntactic analyzer of shell scripts following the traditional design found in most textbooks [2], that is a pipeline of a lexer followed by a parser. Hence, we cannot use either the standard interfaces of code generated by LEX and YACC, because these interfaces have been designed to fit this traditional design.

In this situation, one could give up using code generators and fall back to the implementation of a hand-written character-level parser. This is done in DASH for instance: the parser of DASH 0.5.7 is made of 1569 hand-crafted lines of C code. This parser is hard to understand because it is implemented by low-level mechanisms that are difficult to relate to the high-level specification of the POSIX standard: for example, lexing functions are implemented by means of **goto**s and complex character-level manipulations; the parsing state is encoded using activation and deactivation of bit fields in one global variable; some speculative parsing is done by allowing the parser to read the input tokens several times, etc.

Other implementations, like the parser of BASH, are based on a YACC grammar extended with some code to work around the specificities of shell parsing. We follow the same approach except on two important points. First, we are stricter than BASH with respect to the POSIX standard: while BASH is using an entirely different grammar from the standard, we literally cut-and-paste the grammar rules of the standard into our implementation to prevent any change in the recognized language. Second, in BASH, the amount of hand-written code that is accompanying the YACC grammar is far from being minimal. Indeed, we counted approximately 5000 extra lines of C to handle the shell syntactic peculiarities. Our implementation only needed approximately 250 lines of OCaml to deal with them.

Of course, these numbers should be taken with some precaution since OCaml has a higher abstraction level than C, and since BASH implements a large extension of the shell language. Nonetheless, we believe that our design choices greatly help in reducing the amount of *ad hoc* code accompanying the YACC grammar. The next sections try to give a glimpse of the key aspects of our parser implementation.

A. A modular architecture

Our main design choice is not to give up on modularity. As shown in Figure 3, the architecture of our syntactic analyzer is similar to the common architecture found in textbooks as we clearly separate the lexing phase and the parsing phase in two distinct modules with clear interfaces. Let us now describe the original aspects of this architecture.

As suggested by the illustration, we decompose lexing in two distinct subphases. The first phase called "prelexing" is implementing the "token recognition" process of the POSIX standard. As said earlier, this parsing-independent step classifies the input characters into three categories of "pretokens": operators, words and potentially significant layout characters (newline characters and end-of-input markers). This module is implemented using OCAMLLEX, a lexer generator distributed with the OCAML language. In Section III-B, we explain which features of this generator we use to get a high-level implementation of lexical conventions close to the informal description of the specification.

The second phase of lexing is parsing-dependent. As a result, a bidirectional communication between the lexer and the parser is needed. On one side, the parser is waiting for a stream of tokens to reconstruct a parse tree. On the other side, the lexer needs some parsing context to promote words to keywords or assignment words, to switch to the lexing mode for here-documents, and to discriminate between the four interpretations of the newline character. We manage to implement all these *ad hoc* behaviors using speculative parsing, which is easily implemented thanks to the incremental and purely functional interface produced by the parser generator MENHIR[16]. This technique is described in Section III-C.

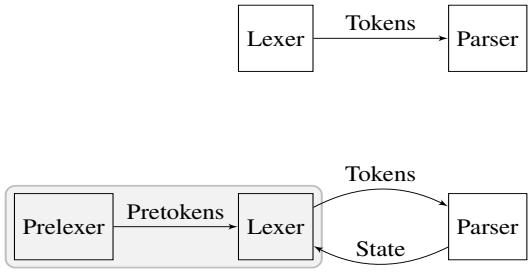

Fig. 3. Architectures of syntactic analyzers: at the top of the figure, the standard pipeline commonly found in compilers and interpreters ; at the bottom of the figure, the architecture of our parser in which there is a bidirectional communication between the lexer and the parser.

B. Mutually recursive parametric lexers

The lexer generators of the LEX family are standard tools that compile a pattern matching made of regular expressions into an efficient finite state machine. When a specific regular expressions is matched, the generated code triggers the execution of an arbitrary piece of user-written code. In theory, there is no limitation on the computational expressiveness of lexers generated by LEX since any side-effect on the lexing engine can be performed in the arbitrary code attached to each regular expression. In practice though, it can be difficult to develop complex lexical analyzers with LEX especially when several sublexers must be composed to recognize a single token which is the concatenation of several words of distinct nature (like the word `$BAR" "$(echo $(date))` we encountered earlier) or when they have to deal with nested constructions (like the parenthesized quotations of the shell language for instance).

OCAMLLEX is the lexer generator of the OCAML programming language. OCAMLLEX extends the specification language of LEX with many features, two of which are exploited in our implementation.

First, in OCAMLLEX, a lexer can be defined by a set of mutually recursive entry points. This way, even if a word can be recognized as a concatenation of words from distinct sublanguages, we are not forced to define these sublanguages in the same pattern matching: on the contrary, each category can have a different entry point in the lexer which leads to modular and readable code. Thanks to this organization of the lexical rules, we were able to separate the lexer into a set of entry points where each entry point refers to a specific part of the POSIX standard. This structure of the source code

eases documentation and code reviewing, hence it increases its reliability.

Second, each entry point of the lexer can be parameterized by one or several arguments. These arguments are typically used to have the lexer track contextual information along the recognition process. Combined with recursion, these arguments provide to lexers the same expressiveness as deterministic pushdown automata. This extra expressive power of the language allows our lexer to parse nested structures (e.g. parenthesized quotations) even if they are not regular languages. In addition, the parameters of the lexer entry points make it possible for several lexical rules to be factorized out in a single entry point. For instance, we factorized the recognition of nested backquotes, parenthesizes and braces using a single entry "`next_nesting nestop level current`" where `nestop` is the kind of nesting that is being recognized.

C. Incremental and purely functional parsing using Menhir

YACC-generated parsers usually provide an all-or-nothing interface: when they are run, they either succeed and produce a semantic value, or they fail if a syntax error is detected. Once called, these parsers take the control and do not give it back unless they have finished their computation. During its execution, a parser calls its lexer to get the next token but the parser does not transmit any information during that call since lexing is usually independent from parsing.

As we have seen, in the case of the shell language, when the lexer needs to know if a word must be promoted to a keyword or not, it must inspect the parser context to determine if this keyword is an acceptable token at the current position of the input. Therefore, the conventional calling conventions of lexers from parsers is not adapted to this situation.

Fortunately, the MENHIR parser generator has been recently extended by François Pottier to produce an incremental interface instead of the conventional all-or-nothing interface. In that new setting, the caller of a parser must manually provide the input information needed by this parser for its next step of execution and the parser gives back the control to its caller after the execution of this single step. Hence, the caller can implement a specific communication protocol between the lexer and the parser. In particular, the state of the parser can be transmitted to the lexer.

Now that the lexer has access to the state of the parser, how can it exploit this state? Must it go into the internals of LR parsing to decipher the meaning of the stack of the pushdown automaton? Actually, a far simpler answer can be implemented: the lexer can simply perform some speculative parsing to observationally deduce information about the parsing state. In other words, to determine if a token is compatible with the current parsing state, the lexer just executes the parser with the considered token to check whether it produces a syntax error, or not. If a syntax error is raised, the lexer backtracks to the parsing state before the occurrence of this speculative parsing.

If the parsing engine of MENHIR were imperative, the backtracking process required to implement speculative parsing would imply some machinery to undo parsing side-effects. This kind of machinery is not needed because the parsing engine of MENHIR is purely functional: the state of the parser is an explicit value passed to the parsing engine which returns in exchange a fresh new parsing state without modifying the input state. From the programming point of view, backtracking is as free as declaring a variable to hold the state to recover to if a speculative parsing goes wrong.

D. Benchmarks

On a i7-4600U CPU @ 2.10GHz with 4 cores, an SSD hard drive and 8GB of RAM, the average time to parse a script is 12ms (with a standard deviation which is less than 1% of this duration). The maximum parsing time is 100ms, reached for the `prerm` script of package `w3c-sgml-lib_1.3-1_all` which is 1121 lines long.

Using 4 workers dispatched over the 4 cores, it takes 61s to parse the 31521 scripts of the corpus. To avoid reparsing the scripts for each analysis, the concrete syntax trees are serialized on the disk. Loading all these files in memory takes only 120ms.

IV. ANALYZER

Once we have a way to parse (most) shell scripts into concrete syntax trees, we need a way to define different analyzers on these concrete syntax trees.

The difficulty of writing analyzers lies in the number of different syntactic constructions of a realistic language like shell: the concrete syntax trees have 108 distinct kinds of node. Even if most of the time a single analysis focuses on a limited number of kinds of nodes, the analyzer must at least traverse the other kinds of node to reach them.

This problem is well-known in software engineering and it enjoys a well-known solution as well: the visitor design pattern [9]. We follow a slightly modified version of this design pattern. A visitor for our concrete syntax trees is an object which has a method for each kind of nodes. Such a method expects as arguments the children of the nodes it handles.

Our first visitor is a generic iterator object that only goes through the concrete syntax tree without doing anything. A concise definition of any analysis can be obtained by inheriting from the generic iterator and redefining specific methods for the constructions under the focus of this analysis. Most of our analyses can be run in only one pass through the concrete syntax of each script. An example of a toy analyzer counting the number of **for**-loops in the corpus is given in Figure 4. On line 1, we declare the analyzer as a module that must implement the common interface `Analyzer.S` of analyzers. On line 2, we declare the name for the analyzer and on line 4, the fact that there is no command line option associated to this analyzer. On line 8, the function `process_script` describes how the complete commands of the script must be processed: on line 22, the function simply applies a locally defined visitor `iterator'` on each complete command. Thanks to the inheritance mechanism used on line 12, most of the methods

are reused as-is. The only redefined method appear on line 14 and it describes what must be done on `for` constructions: we simply increment a global counter `count` introduced on line 6 and recurse on the body of the **for**-loop. Finally, the function `output_report ()` implements the final printing of the analysis results.

a) Miscellaneous: We first have a few very small analyzers counting various things like the number of uses of the `$@` or `$*` variables, the number of commands whose name is taken from variables, the number of scripts that manipulate the `$IFS` variable, etc.

b) Variables: We then have an analyzer that aims to determine the variables that are in fact constants. Identifying a variable as a constant allows us to expand that variable statically and, by doing so, to simplify the script. The variables that we identify as constants are the variables that are assigned to only once in a script, and not under a compound command—*i.e.* at top level (see also Section VI).

c) Structures: We also have an analyzer dedicated to counting control structures (e.g. if, case, for, etc.). The default is to just count, but one can then add custom treatments for specific structures.

We have, for instance, a specific treatment of the for construct: we are using this analyzer to determine whether or not for loops can be unfolded, which is possible if the list they are looping through is known statically. This is achieved by determining whether the arguments of for loops use variables, commands or globs.

We also count the number of case instructions that are used to match on the first argument of the script—the `$1` variable. Once again, this is an important piece of information, insofar as it will allow us to ignore irrelevant parts of scripts when working on particular cases.

d) Functions: An other analyzer helps us in classifying uses of functions: in addition to counting them, it also constructs a dependency graph of the function calls in order to detect cycles, which would indicate (mutually) recursive definitions. It also detects when a script defines more than one function with the same name.

e) Commands: Finally, we have an analyzer working specifically on the use of simple commands. This helps us a lot in knowing what commands we should prioritize in our work. It basically gives us three things:

- The number of uses of commands in the corpus;
- The number of scripts that don't use too rare commands;
- The combinations of arguments that are used by packages maintainers.

In order to get a usable result for arguments combination, we had to provide small specifications of the way these arguments may be combined for a specific command. These specifications list the names of the legal options of a command with their aliases and number of arguments. They also tell if a script may contract its options, as in `rm -rf` for `rm -r -f`. In the cases where we have a specification of the legal options of a UNIX command, our tool also gives as a list of unknown options used with a command. This might indicate a bug in the script, or that a variant of an option is not properly documented in the manpage of the command.

Construct	Occurrences	Files
`alias`	2	2
`eval`	42	30

Fig. 5. Constructs which may render analysis impossible

Construct	Occurrences	Files
`if`	56.643	27.122
`while`	4.045	3.929
`until`	1	1
`for`	3.564	2.400
`case`	6.227	5.296

Fig. 6. Sequential control structures

The frequency of the different UNIX commands in the corpus also allows us to estimate how many scripts we would have to discard from our future analysis if we restricted ourselves to scripts using only frequently used commands. We define, for any natural number i, an *exotic command of level* i to be a command that isn't found in more than i scripts. For given levels of exotism, we count the number of commands that are exotic of this level, and—more importantly—the number of scripts that don't use any of these commands.

V. Results

We have analyzed the 31.832 maintainer scripts present in the Debian *unstable* distribution for the AMD64 architecture, in the areas MAIN, CONTRIB, and NON-FREE, as of 29 Nov 2016. 296 of these are `bash` scripts, 14 are `perl` scripts, one is an ELF executable[1], and hence out of scope for us. Our parser succeeds on 31.484, that is 99.88% of the remaining 31.521 POSIX shell scripts.

A. Shell features

The first question we have investigated is which features of the shell are used with which frequency. It should be noted that already this analysis goes beyond what can be done with a simple `grep -c`, due to the difficulties of lexical analysis explained in Section II.

Figure 5 summarizes the occurrences of shell constructs which may render syntactic analysis impossible, as explained in Section II. The `alias` construct appears only twice in our corpus, and both occurrences are at the top level. There are 42 occurrences of `eval` in 30 scripts.

The occurrences of the different sequential command structures of the shell are given in Figure 6. Constructs related to process creation and communication are given in Figure 7. This table shows that the use of `&`, which creates an asynchronous execution, is very rare in maintainer scripts. This observation, together with the fact that `dpkg` does not allow for concurrent execution of maintainer scripts, justifies our decision to ignore concurrency in the modelization of shell

[1]We let the reader find out which package cannot have a `preinst` script written in shell.

```
module Self : Analyzer.S = struct
  let name = "count-for"

  let options = []

  let count = ref 0

  let process_script filename csts =
    let module Counter = struct
      class iterator' = object(self)

        inherit CST.iterator

        method on_for_clause = incr count; CST.(function
          | ForClause_For_Name_LineBreak_DoGroup (_, _, d)
          | ForClause_For_Name_LineBreak_In_SequentialSep_DoGroup (_, _, _, d)
          | ForClause_For_Name_LineBreak_In_WordList_SequentialSep_DoGroup (_, _, _, _, d) ->
            self#on_do_group d)
      end
    end
    in
    List.iter (new Counter.iterator')#on_complete_command csts

  let output_report () =
    Printf.printf "There are %d for-loops.\n" (!count)
end
```

Fig. 4. A toy analyzer counting the number of **for**-loops in a script.

Construct	Occurrences	Files
subshell	431	356
pipe	12.225	6.154
`trap`	32	28
`kill`	39	35
`&`	8	7

Fig. 7. Process creation and communication.

Construct	Occurrences	Files
`set`	30.817	30.579
`exit`	13.915	8.685
`echo`	10.770	5.010
`true`	10.740	3.966
dot	4.922	2.900

Fig. 8. Simple shell builtins

scripts [reference hidden]. The five most frequent simple shell builtins are listed in Figure 8. The dot symbol, which is used to include another file in the shell script, has almost 5.000 occurrences and hence has definitely to be handled by any future formal treatment of shell scripts.

B. Structure of scripts

The construction of the concrete syntax trees allows us to go further than just simple counting of occurrences of reserved words, and do a more *structural* analysis of shell scripts.

One important question for the kind of formal analysis we wish to perform in the future on scripts is how to handle loops. As we have seen in Figure 6, almost half of the loops found in our corpus are `for` loops. Is it possible to statically unfold these loops? Our analysis shows indeed that only 1.147 of the 3.564 `for` loops iterate over the value of an expression

```
for pyversion in 2.4 2.5; do
  if [ -d /usr/lib/python$pyversion/s/m/ ]; then
    rm -fr /usr/lib/python$pyversion/s/numpy*
  fi
done
```

Fig. 9. A `for` loop which can be unrolled statically (directory names are abbreviated).

containing variables, 213 contain a subshell invocation (`$(·)`, or backquotes), and 126 contain a glob which is subject to filename expansion. In other words, at least 59% of the for loops can be statically unrolled. A typical example is shown in Figure 9.

Another question concerns the use of `case`. It turns out that 5.062 of the 6.227 occurrences of `case`, that is more than 80%, do a matching of the expression `$1`, that is the first argument of the invocation of the script. The Debian policy manual [3], which governs our use case, defines which are the possible first arguments with which a maintainer scripts may be invoked[2]. This means that we will be able to unfold, in any actual verification task, any such `case` statement.

A significant portion of the variables defined in maintainer scripts are in fact constants: We found that in 1295 out of 2841 cases (33%), a shell variable is in fact a constant, that is it is assigned to once in the script, and this assignment occurs at top level (see also the discussion in Section VI).

Function definitions are quite frequently used in maintainer scripts: we found 3.455 function definitions in 1.500 files.

[2]these are strings describing the action, like `install` or `upgrade`

Command	Occurrences	Command	Occurrences
[	47.633	find	2.144
which	12.669	xargs	1.907
rm	10.383	test	1.594
grep	5.138	chmod	1.562
read	3.896	chown	1.504

Fig. 10. The ten mostly used UNIX commands acting on the file system

Options	Occurrences	Options	Occurrences
-s	333	*(none)*	5
-f -s	210	-f	4
-r -s	31	-S *arg0* -b -s	4
-f -n -s	10	-b -f -s	3
-s -v	5		

Fig. 11. Options of ln (605 invocations in total)

Only one single function definition is recursive[3]. This means that we can, with only one exception, unroll function definitions in scripts, and hence may ignore functions in our verification tools. We also found 9 maintainer scripts which contain multiple definitions of the same function. Four of these cases are caused by a tool of the DEBHELPER family that should be optimized. There are four scripts which define the same function differently in the two branches of an if-then-else, which is in our opinion perfectly OK, and one script containing two *slightly* different definitions for the same function, which could be improved by factorizing the large common part of the two definitions.

C. UNIX commands

Our tool provides a statistics of the number of occurrences of each possible combination of options. Figure 11, for example, yields the combination of options observed for the `ln` command, together with their number of occurrences. One important conclusion for us is that 596 out of the 605 invocations of `ln` create symbolic links instead of hard links. The possibility of multiple hard links in a file system are a problem for any formal model of file systems since it means that one has to use acyclic directed graphs as a model, instead of the much simpler trees. The fact that the creation of multiple hard links (`ln` without the `-s` option) is rather rare justifies our decision to consider file systems as trees, at least in a first approach. Our specification of the `ln` command indicates that the `-S` option takes one argument, which is displayed in Figure 11 as *arg0*.

[3]the function `run_command` in the `postinst` script of the package `rt4-extension-assettracker`, version `3.0.0-1`

Command	Occurrences	Files
dpkg-maintscript-helper	9.992	3.889
dpkg	6.862	6.518
deb-systemd-helper	4.530	1.029
update-alternatives	3.616	2.350
update-menus	3.363	3.336

Fig. 12. Top 5 Debian-specific commands

Level	Number	Percentage
1	693	2.20%
2	1.032	3.28%
5	1.459	4.63%
10	1.794	5.70%
25	2.364	7.51%
50	3.286	10.44%
100	4.058	12.89%
200	5.232	16.62%
500	8.095	25.71%

Fig. 13. Number of scripts using exotic commands

Figure 12 yields the 5 most frequently used Debian-specific commands in our corpus. These commands are much harder to model than the standard UNIX commands since they typically manipulate the contents of files. The statistics on command usage help us to focus on the most important ones among these complex commands.

We have filed a number of Debian bug reports[4] against individual packages, or against Debian tools when appropriate.

Finally, Figure 13 tells how many scripts use exotic commands. For instance, 1.794 scripts use at least one command that occurs in at most 10 scripts (see Section IV).

D. Reproducibility

A corpus of maintainer scripts as the one we used for our analysis can easily be extracted on any mirror of the Debian archive, by extracting the maintainer scripts of all packages of the chosen distribution and areas using `dpkg-deb -e`. One should keep in mind, however, that the contents of Debian *sid* is updated four times a day, so that there will certainly be differences to the version we used. Also, if you have to download the complete packages first, instead of working directly on an archive mirror, you should be prepared to download about 200GB of packages.

VI. Current limitations and future work

An important issue is how to validate our parser. Counting the number of scripts that are recognized as being syntactically correct is only a first step since it does not tell us whether the syntax tree constructed by the parser is the correct one. We can imagine several ways how the parser can be validated.

One approach is to write a *pretty-printer* which sequentializes the concrete syntax tree constructed by the parser. The problem is that our parser has dropped part of the layout present in the shell script, in particular information about spaces, and comments. Still, a pretty printer can be useful to a human when verifying the correct action of the parser on a particular case of doubt. It might also be possible to compare the result obtained by our pretty-printer with the original script after passing both through a simple filter that removes comments and normalizes spaces. Furthermore, a pretty-printing functionality can be used for an automatic smoke test on the complete corpus: the action which consists

[4]https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org

of parsing a shell script and then pretty-printing it must be idempotent, that is performing it twice on a shell script must yield the same result as performing it once.

Another possible approach is to combine our parser with an interpreter that executes the concrete syntax tree. This way, we can compare the result of executing a script obtained by our interpreter with the result obtained by one of the existing POSIX shell interpreters.

Our current variable analyzer is certainly too simple, and just serves as a proof of concept. We will have to use a more sophisticated flow analysis of the code in order to detect the values of which variables at which locations in the code can be statically deduced.

VII. Related work

To our knowledge, the only existing attempt to mine a complete corpus of package maintainer scripts was done in the context of the Mancoosi project [6]. An architecture of a software package installer is proposed that simulates a package installation on a model of the current system in order to detect possible failures. The authors have identified 52 templates which cover completely 64.3% of all the 25.440 maintainer scripts of the Debian Lenny release. These templates are then used as building blocks of a DSL that abstracts maintainer scripts. In this work, a first set of script templates had been extracted from the relevant Debian toolset (DEBHELPER), and then extended by clustering scripts using the same statements [8]. The tool used in this works is geared towards comparing shell scripts with existing snippets of shell scripts, and is based on purely textual comparisons.

There have been few attempts to formalize the shell. Recently, Greenberg [10] has presented elements of formal semantics of POSIX shell. The work behind Abash [14] contains a formalization of the part of the semantics concerned with variable expansion and word splitting. The Abash tool itself performs abstract interpretation to analyze possible arguments passed by Bash scripts to UNIX commands, and thus to identify security vulnerabilities in Bash scripts.

Some of the problems with POSIX shell are also encountered in other scripting languages, For instance, [18] uses an existing parser to analyze PHP scripts into abstract syntax trees, and then performs symbolic execution in order to detect security vulnerabilities.

Several tools can spot certain kinds of errors in shell scripts. The CHECKBASHISMS [5] script detects usage of Bash-specific syntax in shell scripts, it is based on matching Perl regular expressions against a normalized shell script text. This tool is currently used in Debian as part of the `lintian` package analyzing suite. The tool SHELLCHECK [11] detects error-prone usage of the shell language. This tool is written in Haskell with the parser combinator library PARSEC. Therefore, there is no `Yacc` grammar in the source code to help us determine how far from the POSIX standard the language recognized by SHELLCHECK is. Besides, the tool does not produce intermediate concrete syntax trees which forces the analyses to be done on-the-fly during parsing itself. This approach lacks modularity since the integration of any new analysis requires the modification of the parser source code. Nevertheless, as it is hand-crafted, the parser of SHELLCHECK can keep a fine control on the parsing context: this allows for the generation of very precise and helpful error messages. We plan to use the recent new ability [15] of MENHIR to obtain error messages of similar quality.

VIII. Conclusion

Statically parsing shell scripts is notoriously difficult, due to the fact that the shell language was not designed with static analysis, or even compilation, in mind. Nevertheless, we found ourselves in need of a tool that allows us to perform easily a number of different statistical analyses on a large number of scripts. We have written a parser that maintains a high level of modularity, despite the fact that the syntactic analysis of shell scripts requires an interaction between lexing and parsing that defies traditional compiler design. The definition of the resulting concrete syntax tree as an object allowed us to easily define different analyzers based on a visitor design pattern. We have already obtained many useful results that are guiding us in the design of the formal verification tools we are going to build.

Acknowledgment

We thank Patricio Pelliccione and Davide Di Ruscio for discussion of their work done in the context of the Mancoosi project.

References

[1] P. Abate, R. D. Cosmo, L. Gesbert, F. L. Fessant, R. Treinen, and S. Zacchiroli. Mining component repositories for installability issues. In M. D. Penta, M. Pinzger, and R. Robbes, editors, *MSR 2015*, pages 24–33, Florence, Italy, May 2015. IEEE.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] R. Allbery, B. Allombert, A. Barth, and J. Nieder. Debian policy manual, Mar. 2016. http://www.debian.org/doc/debian-policy/.

[4] C. Artho, K. Suzaki, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In M. Lanza, M. D. Penta, and T. Xie, editors, *MSR 2012*, pages 141–150, Zurich, Switzerland, June 2012. IEEE Computer Society.

[5] R. Braakman, J. Rodin, J. Gilbey, and M. Hobley. checkbashisms. https://sourceforge.net/projects/checkbaskisms/, Nov. 2015.

[6] R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming*, 76(12):1144–1160, 2011.

[7] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon. Maintaining large software distributions: new challenges from the FOSS era. In *Proceedings of the FRCSS 2006 workshop*, volume 12 of *EASST Newsletter*, pages 7–20, 2006.

[8] D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Towards maintainer script modernization in FOSS distributions. In *IWOCE 2009: International Workshop on Open Component Ecosystem*, pages 11–20. ACM, 2009.

[9] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[10] M. Greenberg. Understanding the POSIX shell as a programming language. In *Off the Beaten Track 2017*, Paris, France, Jan. 2017.

[11] V. Holen. shellcheck. https://github.com/koalaman/shellcheck, 2015.

[12] IEEE and The Open Group. The open group base specifications issue 7. http://www.unix.org/version3/online.html, 2016.

[13] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, pages 199–208, Tokyo, Japan, Sept. 2006. IEEE CS Press.

[14] K. Mazurak and S. Zdancewic. ABASH: finding bugs in bash scripts. In *PLAS07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 105–114, San Diego, CA, USA, June 2007.

[15] F. Pottier. Reachability and error diagnosis in LR(1) parsers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 88–98, 2016.

[16] F. Pottier and Y. Régis-Gianas. The Menhir parser generator. *See: http://gallium. inria. fr/fpottier/menhir*.

[17] J. Signoles and S. Boldo, editors. *18ièmes Journées Francophones des Langages Applicatifs*, Gourette, France, Jan. 2017.

[18] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In A. D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, July 2006. USENIX Association.